Skeletons for parallel image processing: an overview of the SKiPPER project

Jocelyn Sérot a,* Dominique Ginhac b

aLASMEA, UMR 6602 CNRS, University Blaise Pascal de Clermont-Ferrand, Campus des Cézeaux, F-63177 Aubière, France. E-mail: Jocelyn.Serot@lasmea.univ-bpclermont.fr

bLE2I, FRE 2309 CNRS, University of Burgundy, F-21078 Dijon, France. E-mail: dginhac@u-bourgogne.fr

Abstract

This paper is a general overview of the SKiPPER project, led at Blaise Pascal University between 1996 and 2002. The main goal of the SKiPPER project was to demonstrate the applicability of skeleton-based parallel programming techniques to the fast prototyping of reactive vision applications. This project has produced several versions of a full-fledged integrated parallel programming environment (PPE). These PPEs have been used to implement realistic vision applications, such as road following or vehicle tracking for assisted driving, on embedded parallel platforms embarked on semi-autonomous vehicles. All versions of SKiPPER PPEs share a common front-end and repertoire of skeletons—presented in previous papers—but differ in the techniques used for implementing skeletons. This paper focuses on these implementation issues, by making a comparative survey, according to a set of four criteria (efficiency, expressivity, portability, predictability), of these implementation techniques. It also gives an account of the lessons we have learned, both when dealing with these implementation issues and when using the resulting tools for prototyping vision applications.

Key words: Parallelism, skeleton, computer vision, fast prototyping, data-flow

1 Introduction

The general context of the SKiPPER project is the development of realistic vision applications for embedded platforms. These applications may be found...
for instance in remote inspecting robots or vehicles equipped with assisted-driving systems, as presented in [25], [27]. Although relying on algorithms and programming paradigms encountered in the mainstream of computer vision, these applications raise two specific issues. First, they implement reactive systems, operating “on the fly” on digital streams of images. This means that they must be able to absorb input data and output results at a minimum frequency and produce responses within a maximal latency. For assisted-driving applications, for instance, the typical frequencies are in the range of 10-30 frame/s and the maximal latency rarely exceeds 50 ms. Second, they must meet stringent operational constraints in terms of volume or power consumption, which often rules out implementations based upon stock-hardware.

We found that these requirements can be met by resorting to embedded parallel machines. The TRANSVISION [19,15] platforms, built between 1992 and 1998 at LASMEA are examples of this approach. These MIMD architectures, built upon Transputer and Alpha processors could deliver significant computing power and provided built-in facilities for video i/o. More recently, we have been investigating the feasibility of a Beowulf-style cluster built upon PowerPC G4 processors and using the IEEE-1394 interface for fast video i/o.

But relying on parallel machines place severe strains on programmers: in the absence of high-level parallel programming models and environments, they have to explicitly take into account every aspects of parallelism such as task partitioning and mapping, data distribution, communication scheduling or load-balancing. Having to deal with these low-level details results in long, tedious and error-prone development cycles – especially when the persons in charge of developing the algorithms are image processing, not parallel programming, specialists –, thus hindering a true experimental approach. For reactive applications, the problem is reinforced by the fact that the need to evaluate the dynamic properties of the algorithm at realistic frame-rate effectively rules out any prototyping phase solely based upon off-line, sequential simulation on stock hardware. Parallel programming at a low level of abstraction also limits code reusability and portability.

The SKIPPER project was developed in response of the aforementioned problems. Basically, its goal was to “capture” – in a efficient and portable way – the expertise gained by programmers when implementing reactive vision applications using low level parallel constructs, to make it readily available to algorithmicians and image processing specialists. This project has been led at LASMEA between from 1996 to now and has produced four skeleton-based parallel programming environments : SKIPPER [-o], SKIPPER [-i], SKIPPER [-ii] and SKIPPER [-d]. These realisations have been described in previous papers [16,25,27,9,24] but in a rather independent manner. The goal of this paper is to fill the gap between these separate accounts and to provide a comparative assessment of the successive versions of SKIPPER. It explains in particular
why these versions, which share a common formalism for specifying parallel programs, differ significantly in the techniques used for implementing skeletons. It is organized as follows. Section 2 is a brief recall of SKIPPER principles and general architecture. The discriminating features of SKIPPER versions and the criteria used to assess them will be highlighted in Section 3. Each version of SKIPPER will be presented in turn in Sections 3.1, 3.2, 3.3 and 3.4. Section 4 concludes this paper by ...

2 SKIPPER generic architecture

The SKIPPER programming methodology is based upon the concept of algorithmic skeletons [7,8]. Skeletons are high-level program constructs that abstract common patterns of parallel computation in a parametric way. With this approach, the structure of a parallel application is expressed only as a combination of the skeletons provided. The repertoire of skeletons acts as a sort of "parallel toolbox" from which parallel programs can be built with a minimal concern for low-level details. This is illustrated in Fig. 1, which gives the general architecture of SKIPPER. The application programmer provides a skeletal, structured description of the parallel program, the set of application-specific sequential functions used to instantiate the skeletons and a description of the target architecture. SKIPPER provides tools from turning these descriptions into executable parallel code. The main software components are: a library of skeletons, a compile-time system (CTS) for generating the parallel C code and a run-time system (RTS) providing support for this executing this parallel code on the target platform. The CTS can be further decomposed into a front-end, whose goal is to generate a target-independent intermediate representation of the parallel program, and a back-end system, in charge of mapping this intermediate representation onto the target architecture.

The skeleton library. The SKIPPER library of skeletons was built "bottom-up", from a careful analysis of a large corpus of existing low-to-mid level vision applications hand-coded in parallel C[27]. It consists of three skeletons:\footnote{A fourth skeleton (itermem) is described in some previous papers. This skeleton does not properly speaking encapsulate parallel behaviour, but is used whenever the iterative nature of the real-time vision algorithms – i.e. the fact that they do not process single images but continuous streams of images – has to be made explicit. It will not be discussed here.}

- the scm (split-compute-merge) skeleton is devoted to fixed data-parallelism, for instance to "geometric" processing of iconic data, in which the input image is split into a fixed number of subimages, each subimage is processed

\[ \text{scm} \]

\[ \text{itermem} \]
independently and the final result is obtained by merging the results computed on subimages;

- the \texttt{df} (data-farming) skeleton handles variable data-parallelism, i.e. situations in which the number of data to process is not known \textit{a priori};
- the \texttt{tf} skeleton is a generalisation of the \texttt{df} one, in which the processing of one data item may recursively generate new items to be processed. It is classically used to implement \textit{divide-and-conquer} strategies.

Each skeleton comes with two semantics: a \textit{declarative semantics}, which gives its “meaning” to the application programmer in an implicitly parallel manner, i.e. without any reference to an underlying execution model, and an \textit{operational semantics} which provides an explicitly parallel description of the skeleton.

The declarative semantics of each skeleton is shared by all \textsc{skipper} versions. It is conveyed using the CAML language, using higher-order polymorphic functions. The corresponding definitions are given in Fig. 2. Potential (implicit) parallelism arises from the use of the \texttt{map} and \texttt{foldl1} higher-order functions\footnote{These higher-order functions applies a function and iterates an (commutative, associative) binary operator over a list of elements, respectively: \texttt{map} \( f \) \[ [x_1, x_2, \ldots, x_n] \] = \[ f(x_1), f(x_2), \ldots, f(x_n) \] and \texttt{foldl1} \( \oplus \) \[ [x_1, x_2, \ldots, x_n] \] = \( x_1 \oplus x_2 \oplus \ldots \oplus x_n. \)}.

The operational semantics of a skeleton varies according to the nature of the

Fig. 1. General software architecture
let scm split comp merge x = merge (map comp (split x))
let df comp acc xs = foldl1 acc (map comp xs)
let rec tf h solve divide acc xs =
  let f x =
    if (h x) then acc z (solve x)
    else tf n h solve divide acc (divide x)
in
foldl1 acc (map f xs)

Fig. 2. Declarative semantics of Skipper skeletons

intermediate representation used by the CTS. In the successive versions of
Skipper, we have been experimenting with four types of intermediate rep-resentations: synchronous data-flow graphs (for Skipper [-o]), parametric pro cess networks (for Skipper [-i]), hierarchical task graphs (for Skipper [-ii])
and tagged-token data-flow graphs (for Skipper [-d]). These representations
will be discussed in turn in sections 3.1 to 3.4.

Skipper also relies on the Caml language for expressing the parallel (skeletal)
structure of the programs. The programmer here indicates which skeletons are
used, in what order and, for each skeleton, the sequential functions and/or
numeric values passed as parameters. This is illustrated in Fig. 3a, with a
small program making use of two SCM skeletons to binarize an image. Here
row_block, histo, merge_histo, get_img, bimod, binar and display_img are
the application specific, sequential functions: row_block decomposes an image
into horizontal sub-images, histo computes the histogram of a (sub)image and
merge_histo sums the partial histograms computed on each subimage into
the final one. bimod and binar respectively computes and applies an optimal
binarization threshold. The get_img and display_img functions respectively
retrieves the next image from the video input stream and displays the binarized
image on the screen.

let src = get_img 256;;
let h = scm row_block histo merge_histo src;;
let th = bimod h;;
let res = scm row_block (binar th) block_row src;;
let main = display_img res;;

Fig. 3. Sample program

In the previous example, the application-specific sequential functions
are written in C. This point is of great practical importance since we don’t
want application programmers to recode their algorithms from scratch (and

---

3 This need to pass and return functions and values from various types to/from
other functions explains the choice of a higher-order, polymorphic language as Caml
for specifying skeletons and skeletal programs in Skipper.

5
The role of the **back-end** in the CTS is to map the intermediate representation of the parallel program (data-flow graph, process network, etc.) onto the target architecture. For a MIMD target with distributed memory, for example, this involves finding a distribution of the operations/processes on the processors and a scheduling of the resultant communications on the provided medium (bus, point-to-point links, etc.). The distribution and the scheduling can be *static* — i.e., done at compile time — or *dynamic* — i.e., postponed until run-time. Both approaches require some kind of run-time support (RTS). For static approaches, this RTS can be very small since it basically only has to provide mechanisms for synchronizing threads of computations and exchanging messages between processors\(^4\). For this reason, they generally offer more predictable and often better performances. But they may lack expressivity, as evidenced in Sec. 3.1. Dynamic approaches, on the other hand, do not suffer from this limitation (as evidenced in Sec. 3.3 and 3.4) but require bigger and more complex RTS, which often compromises performance predictability and level. The Skipper project has covered a wide spectrum of distribution and scheduling (DS) techniques, ranging from entirely static to fully dynamic, making it possible to assess the relative merits and flaws of these techniques in the context of a skeleton-based methodology.

Depending on the DS technique used in the back-end, the **final parallel code** takes the form of either MPMD (one distinct program per processor) or SPMD (the same program for all processors) C code. This code is linked with the code of the RTS.

### 3 Comparative assessment

All version of Skipper share the general architecture described in the previous section. They differ in the type of *intermediate representation* produced by the front-end and in the distribution/scheduling technique used by the back-end. The following sections make a survey of these features and assess the consequences of each design choice according to four criteria:

**Efficiency.** It will be assessed by comparing the run-time performances of the “skeletized” application with the ones obtained with a carefully hand-crafted parallel version (using C+MPI for instance). It clearly depends on the

---

\(^4\) This can be done in the form of a customizable *kernel* of primitives (like in Skipper \([-o]\)) for example, or by relying on an existing message-passing system like MPI (like in Skipper \([-ii]\).
overhead introduced by this RTS.

**Portability.** Here, we mean the ability to port a given version of the SKIPPER suite of tools on a new parallel platform. Given the layered software architecture of SKIPPER, these portability issues mainly concern the RTS: the smaller (and the simpler) this RTS will be, the more portable the corresponding SKIPPER version will be. Moreover, in our context (embedded vision applications), we must eventually consider the possibility to target architectures with little or no OS-level support\(^5\), such as machines built from specialized or digital signal processors (DSPs).

**Predictability** of performances. This refers to the possibility to predict the run-time behaviour of an application (its latency and frequency for example) without actually running it on the target parallel platform, on the basis of application-specific parameters (such as the duration of the sequential functions) and architecture-specific parameters (such as communication latency). Performance prediction is generally carried out using analytical cost models and estimated (typical) durations (like in SKIPPER [-i] or in most of existing skeleton-based PPEs [3,6,4,23,21]). In the context of reactive applications, one may need a more deterministic approach, in which strict temporal bounds can be computed at compile-time.

**Expressivity.** This refers to the ability to implement an application expressed as an arbitrary combination of skeletons. In practice, experience has shown that the critical point here is whether the intermediate representation supports nesting or not, ie. the ability for a skeleton to take another skeleton as argument. Although it is still unclear whether realistic applications really need nesting (see [8]), its support has always been perceived has a challenge by skeletons’ implementors.

### 3.1 Static data-flow. Skipper-0

The first version of SKIPPER used an intermediate representation of skeletal programs as synchronous data-flow graphs (SDFG) [5]. Skeletons were viewed as parameterisable data-flow graph patterns, encoded directly in CAML as higher-order functions thanks to a tool called CAMLFLOW. An in-depth description of CAMLFLOW (which is based upon abstract interpretation) can be found in [26]. The mapping of the SDFG onto the target architecture was handled by a third party software called SYNDEx [17]. Both the distribution of the operations (the sequential functions associated with nodes) onto the proces-

---

\(^5\) By OS-level support, we mean the facilities typically provided by multi-tasking, Unix-like, operating systems: multi-processing, inter-process communication and synchronization, virtual memory, etc.
sors and the scheduling of communications onto inter-processor channels were static. The result of this DS step was a set of processor-independent programs (\texttt{m4} macro-code, one per processor) built from a small kernel of primitives and offering support for static thread creation, thread communication and call of user-supplied sequential functions. The final parallel C code was obtained by simply providing definitions for these kernel primitives according to the available hardware facilities. The complete compilation path for \textsc{Skipper} [-o] is illustrated in Fig. 4.

Fig. 4. Software architecture in \textsc{Skipper} [-o]

The SDFG/SYNDEX approach is illustrated in Fig. ???. The left window shows the data-flow graph of a simple application making use of the SCM (split, compute then merge) skeleton along with the target architecture (four ring-interconnected processors) on which it must be implemented. The right window illustrates the mapping of operations onto processors computed by SYNDEX, i.e., the distribution of operations onto processors (one per column) and the scheduling of operations (oval boxes) and communications (diagonal lines) on each processor. USE SAME PROGRAM as in Sec 2 !!!!!

![SDFG representation of a skeletal program as handled by the SYNDEX tool](image)

Fig. 5. SDFG representation of a skeletal program as handled by the SYNDEX tool

Assessment. With \textsc{Skipper} [-o], the overhead of the run-time system was virtually zero, since all decisions regarding distribution and scheduling have been taken at compile-time. This resulted in a high efficiency, as evidenced in Fig. ?? which give the total latency and relative speedup for an application computing histograms of gray level images and implemented on a multi-T800 machine.

\footnote{For the Transvision platforms, for example, the primitives used the built-in process switching and channel i/o of the Transputer. But the kernel can be easily ported to other systems, for instance Unix/Linux-based multi-processors communicating trough TCP/IP sockets or MPI.}

\footnote{At the time \textsc{Skipper} [-o] was designed, this platform was the only available.}
Fig. 6. Skipper [-o] performance figures (scm skeleton)

For the same reason, predictability was very good, with measured performances never differing from the predicted ones by more than 5%. These predictions were actually obtained by using a two-pass process: in a first pass, rough estimates of the durations of the sequential functions were given to SYNDEx, which generated a first, sub-optimal, parallel program but with automatic profiling instructions inserted in it. This program could then be run on typical data to extract the real durations. These durations were used in turn to get the final program by means of a mapping and scheduling heuristic based upon minimization of the total latency. One could also use upper bounds for function durations in order to predict worst-case behavior, in order to satisfy hard real-time constraints for instance. Finally, portability was also good: because the output macro-code was built on a small set of kernel primitives, re-targeting an application on an architecture built from a new processor type only required (re)writing this set of kernel primitives. This proved to be a straightforward task for the platform we had to deal with.

The main problem with Skipper [-o] was expressivity. Indeed, giving an operational semantics to the df and tf skeletons in terms of SDFG was problematic. Consider the df skeleton, for instance. This skeleton is used to apply a function to a list of data items when the size of the list is unknown and/or the time to process one item can vary significantly. In this case, a static allocation of the operations (items) to processors is not always possible and would result, anyway, to an uneven work-load between processors (which in turn results in a poor efficiency). The classical solution is therefore to give the operational semantics of the df skeleton as a process network and to rely on

---

8 To our knowledge, Skipper [-o] is the only realization of a skeleton-based PPE capable of handling such hard real-time timing constraints.
9 The kernel definition for the Transputer processor was less than 300 lines of m4 code. Kernels have been written for several well-know DSPs and also for clusters of Unix machines running TCP/IP communication layers.
10 This situation is frequent is reactive vision, where a varying number of region of interest, of varying size, often have to be processed in each frame.
a farming protocol to ensure load-balancing: a master process dynamically does out items to a pool of worker processes and collecting results back, on a “first done, first served” basis. This model, however, cannot be implemented using a static mechanism, in which all communications must be scheduled at compile-time.

3.2 Template-based implementation. Skipper-1

In Skipper-[i], the limitations of Skipper [-o] were overcome by relying on process networks for the intermediate representation of skeletal programs and on implementation templates for skeletons. This approach is the most widely used for existing skeleton-based PPEs ([3,6,23,21]). Implementation templates are “known parametric parallel process networks that efficiently implements a skeleton on a particular parallel target architecture at hand” [13]. They generally take of the form of process graphs that can be parameterized in the parallelism degree (the number of the worker nodes for instance) and the sequential function(s) associated with each node. The intermediate representation of the application as a process network is then obtained by instantiating the skeleton templates¹¹. The most often claimed advantage of template-based approaches is that, being written once and for all for a given architecture, they can be carefully hand-crafted to make them both reliable and highly efficient.

The Camlflow front-end of Skipper was therefore modified to produce process networks out of CAML skeletal descriptions instead of data-flow graphs. For this, each skeleton was described (in CAML, again) as a parametric process network¹². Fig. 6a gives a parametric process network (PPN) for the DF (Data Farming) skeleton¹³. This graph is parametric in the number of worker nodes, in the type of data items exchanged between nodes (denoted with type variables ‘a . . . ‘b) and in the sequential functions run on the nodes farmer and worker (this “parameterization” being denoted with brackets).

The behavior of the farmer and worker processes was stored separately as a parametric process template (PPT). A PPT is a piece of sequential code whose behavior can be specialized by providing numeric parameters, data

¹¹ This instantiation is done on the basis of the provided application-specific sequential functions. It can also take into account some architectural parameters, to adjust the declared parallelism degree of the skeleton to the one actually offered by the architecture for instance.

¹² To facilitate cross-referencing, we use here the terms introduced in [27]. Conceptually, parametric process networks are implementation templates.

¹³ This graph is a simplified one. The PPN actually used in Skipper-[i] appears in Fig. 6b (see later).
The compilation path in SKIPPER [-i] was similar to the one depicted in Fig. 4 for SKIPPER [-o], except that the CAMLFLOW front-end produced a intermediate representation in the form of a process network instead of a data-flow graph\(^{15}\). The back-end tasks were still handled by the SYNDEX software. This may seem contradictory since, as stated in Section 3.1, SYNDEX can only handle synchronous data flow graphs and not process graphs. The solution adopted in SKIPPER-[i] was in fact an hybrid one: process graphs were “viewed” by SYNDEX as data-flow graphs and mapped/scheduled as data-flow graphs. In particular, SYNDEX only scheduled (at compile-time) “static” communications (the ones that mark the start and the end of a farming skeleton for instance). The “dynamic” communications (the ones occurring between the master and the workers during the activity of a farming skeleton) were handled by ad-hoc processes “hidden” in the data-flow nodes. This technique – which amounts to tolerating some “critical sections” of dynamically scheduled code within a globally statically scheduled application – is detailed in [15]. It is illustrated in Fig. 6b, where “static” communications are denoted with plain lines and “dynamic” ones with dashed lines. Synchronization barriers were used to ensure that the dynamic communications did not interfere, at run time, with the static ones.

**Assessment.** The SKIPPER-[i] version was the first to support the complete set of skeletons described in Sec. 2 and has been used to implement several realistic reactive vision applications, most noticeably those described in [16] (segmentation by connected component labeling), [25] (vehicle tracking) and [27] (road tracking). Thanks to the SYNDEX back-end, efficiency remained high (with an overhead never exceeding 25 % for the applications implemented). For applications making use only of “static” skeletons (such as SCM), this overhead was almost zero, as for SKIPPER [-o]. Fig. 8 gives the measured performances of an application doing some processing on a list of windows of

---

\(^{14}\) Specialization is carried out using macro substitution.

\(^{15}\) A detailed presentation of SKIPPER [-i] can be found in [27].
interests (WOIs). Total latency and relative speedup are given for three values of \( nbw \), the number of WOIs processed in each image (the greater the number is, the more the dynamic farming capabilities of the \( df \) skeleton is solicited).

![Fig. 8. Compilation path in the SKIPPER-[i] parallel programming environment](image)

Predictability of performances relied on a set of analytical cost models [15] that provided an accuracy in the range of 10-20%. But, unlike SKIPPER-[o], strict timing bounds could not always be exhibited: this is clearly the price to pay for accepting dynamically scheduled skeletons such as DF.

The main problem with SKIPPER-[i] lied in the hybrid nature of the intermediate representation. Because dynamic communications were transparent to SYDEX, the routing of these communications between distant processors had to be handled explicitly by some auxiliary processes (whereas it is done automatically by SYDEX for static communications). It turned out that including the description of these auxiliary processes to the SYDEX kernel without compromising too much efficiency was a difficult task. To make the problem tractable, the SKIPPER-[i] compilation process therefore made
assumptions on the topology of the target architecture (it had to be ring-interconnected). These assumptions, along with the increased size and complexity of the SYNDEx kernel, lowered the portability of the applications developed with SKIPPer-[i] (compared to SKIPPer-[o]). Finally, the hybrid intermediate representation of SKIPPer-[i] implicitly relied on a “flat” execution model and was definitely not suited for implementing nested skeletons.

3.3 Hierarchical task graphs. Skipper-2

The SKIPPer-[ii] version is based upon an homogeneous intermediate representation of programs as hierarchical task graphs. This design choice was made in order to overcome the difficulties raised by hybrid representations (such as the one used in SKIPPer-[ii]) and to solve the problem of skeleton nesting in a systematic way. For this, and at the implementation level, all skeletons of the SKIPPer repertoire are viewed as specialized instances of a generic skeleton, called tf-ii\textsuperscript{16}. The operational semantics of the tf-ii skeleton is basically the one of a task farming skeleton: a master process doles out tasks to a pool of worker (slave) processes, but here a task can be either a sequential function to be computed or another skeleton to be run. The intermediate representation takes the form of a tree of tf-ii skeletons. It is computed by a modified version of the CAMLFLOW front-end, which uses alternate definitions of the SCM, DF and TF skeletons as specialized calls to the tf-ii higher-order function. This step is illustrated in Fig. 9 where a program making using of three SCM skeletons (two of them nested) is turned into a tree of tf-ii descriptors. In this tree, nodes correspond to skeleton control processes and leafs to sequential functions (a detailed presentation of the SKIPPer-[ii] system can be found in [9] or in the forthcoming [10]).

Interpretation of the intermediate representation within SKIPPer-[ii] is done at run-time by a specialized program (the “kernel”) running in SPMD mode on all processors (see Fig. 10). This kernel – written in C – provides dynamic support for three kind of services: concurrent execution of master and worker processes, inter-process communication (using a subset of MPI-conformant routines) and handling of shared resources such as the worker pool. Whenever a skeleton needs to be run, either as a “top-level” node (on the spine of the tf-ii tree) or as a nested instance, a new copy of the kernel is launched on the local processor. This copy acts as the master of the skeleton. It uses the free resources (idle processors) to allocates new workers. When all resources are busy, the execution of worker processes is sequentialized on the processor running the master process.

Assessment. SKIPPer-[ii] was the first version to use a fully dynamic im-

\textsuperscript{16} For Task Farming, version II.
(a) Original program  (b) Intermediate representation as a tree of tf-ii

Fig. 10. Intermediate representation of skeletal programs within SKIPPER [-ii]

Fig. 11. Compilation path in the SKIPPER [-ii] parallel programming environment

plementation mechanism for skeleton-based programs. This has several advantages. First, in terms of expressivity, since arbitrary nesting of skeletons is naturally supported. The introduction of new skeletons is also facilitated, since it only requires giving their translation in terms of tf-ii. Portability remains acceptable since porting applications to new architectures only requires the porting of the run-time kernel. This, in practice, turned to be a relatively straightforward task. The approach used in SKIPPER [-ii] also provides automatic load-balancing, since all mapping and scheduling decisions are taken at run-time, depending on the available physical resources. In the same vein, sequential emulation is obtained “for free” by just running the program on a single processor. As regards efficiency, preliminary experiments [9] on synthetic applications suggest that for applications exhibiting a sufficiently high compute/communication ratio the overhead of the kernel-based
implementation (compared to hand-crafted C+MPI code) can be less 10 %, although this overhead can grow up to 50 % when the communications costs dominates\(^{17}\). Fig. 11 gives performance results for a simple application using two \texttt{scm} nested skeletons.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{image.png}
\caption{\textsc{Skipper} [-ii] performance figures (Nested \texttt{scm} skeleton)}
\end{figure}

But several problems have been identified in the \textsc{Skipper} [-ii] implementation which in practice have limited its utility in our context. First, predictability of performances is very low. It is very difficult, in particular, to exhibit even approximative cost models for a model in which processors can switch from a \textit{master} to \textit{worker} behavior depending only on actual input data (there’s no “fixed” mapping for dynamic skeletons as in \textsc{Skipper}-[i]). Even the interpretation of execution profiles, generated by an instrumented version of the kernel, turned out to be far from trivial. This point raises a pragmatical problem within a programming methodology based upon experimental validation of solutions: here, one not only needs to obtain quickly a running prototype, but also to be able to understand why a given prototype exhibit poor runtime performances. By contrast, the profiling facilities offered by \textsc{Skipper}-[i] (and detailed in [27]) were much easier to exploit. Second, shared resources are handled in a centralized manner in \textsc{Skipper} [-ii] (each worker allocation requires a couple of communication to a particular processor, in particular). This centralization can become a bottleneck when the size of the intermediate representation increases. Finally, it turned out that the resource allocation strategy used in \textsc{Skipper} [-ii] only performed well on architectures made of processors supporting \textit{multi-processing}. If not, some processors may end up running only one \textit{master} process, with a small load factor, leading to a poor global efficiency\(^{18}\). This problem has limited in practice the applicability of the \textsc{Skipper} [-ii] system in our context which, in a rather unexpected way, appears more suitable for massively parallel, Beowulf-like clusters than for embedded target architectures.

\(^{17}\) In [9], this is explained by the fact that the kernel intrinsically performs more communications than raw MPI, for exchanging data between inner and outer \textit{masters} in particular.

\(^{18}\) If the \textit{multi-processing} case, the processor can be shared between \textit{master} and \textit{worker} processes.
The implementation of SKIPPER [-d] started in 2000 and was inspired by results obtained by M. Danelutto on the Macro Data-Flow (MDF) execution model for skeletons [12]. This model is very similar to the one used in SKIPPER [-o]: skeleton-based parallel programs are compiled down to data-flow graphs, in which nodes correspond to sequential functions (“macro-instructions”) and arcs to data dependencies between these functions. But, like Danelutto and unlike SKIPPER [-o], a dynamic interpretation mechanism is used for executing these graphs. This mechanism relies on a set of distributed data-flow interpreters, running in SPMD mode on all processors of the target architecture. SKIPPER [-d] extends the MDF execution model proposed by Danelutto in order to implement arbitrary nested data or task farm skeletons. For this, the SKIPPER [-d] runtime relies on the tagged-token data-flow interpretation technique [1,2]. This technique basically allows many concurrent activations of a single sequential node to overlap in time; it associates a unique tag with each activation and each data token also carries a tag that specifies the particular activation to which it belongs. Skeletons involving run-time bounded iterations and/or recursion, and nested in an arbitrary way, can then be represented as cyclic data-flow graphs. This is illustrated in Fig. 12 with the formulation as a tagged-token MDF graph of a program involving two nested df skeletons (in this figure, tags are denoted as superscripts). The MDF graph uses a pair of special nodes called iter and endf. Iter accepts a list of data items and generates distinct result tokens, each carrying one data item and a distinct tag. These tokens trigger distinct firings of the subsequent nodes. The tokens resulting from these firings are collected by the endf and accumulated using the acc sequential function. A more detailed account of this mechanism can be found in [24].

In accordance with the generic software architecture given in Fig. ??, the SKIPPER [-ii] system can be divided into two sub-systems: a run-time system (RTS), implementing a (centralized) tagged-token data-flow interpreter and a compile-time system (CTS), producing the MDF graph for this interpreter from a high-level skeletal program specification.

The run-time system of SKIPPER [-d] is sketched in Fig. 13. Like Danelutto’s system, it relies on an SPMD approach: all the processors (nodes) of the target architecture run the same program, which is the result of the compilation of the user code (C sequential functions) and the interpreter code. The interpreter itself involves several threads of execution: a dispatch thread, which fetches macro-instructions (sequential functions to be computed) from a pool of fireable instructions and sends them to the worker threads, a collect thread, which receives results from the worker threads and updates the instruction
/* Signature of seq. functions */
void getd(intListList *out);
void f(int in, float *out);
void acc2(float in1, float in2, float *out);
void acc1(float in1, float in2, float *out);

(* Program description *)
let xss = getd ()
let inner xs = df f acc1 xs
let outer xss = df inner acc2 xss

Fig. 13. Nested farm skeletons under the tagged-token MDF model

pool accordingly and several\textsuperscript{19} worker threads for computing sequential functions. The dispatch thread fetches idle workers from a centralized pool, in which all worker threads register at initialization and which is subsequently updated by the update thread upon reception of results.

Fig. 14. The run-time system of Skipper [-d].

The compile-time system is sketched in Fig. 14. It produces the application-specific data needed to customize the run-time interpreter, i.e., the MDF representation of the program used to build the initial instruction pool and the

\textsuperscript{19} At least one per processor.
code of the sequential C functions to be integrated with the custom run-time interpreter. The MDF graph is generated by the CAMLFLOW tool. This offers a way, like in previous versions of SKIPPER to describe skeletons entirely in CAML as higher-order functions.

![DIAGRAM]

**Fig. 15.** The run-time system of SKIPPER [-d].

**Assessment.** The main contribution of SKIPPER [-d] is to provide an all-encompassing intermediate representation for all skeletons. This representation allows arbitrary combination (including nesting) of skeletons, thanks to the tagged-token interpretation mechanism. SKIPPER [-d] therefore definitely solves the expressivity problem, at least for our repertoire of skeletons. Experimental results, obtained with a prototype run-time system (written in OBJECTIVE CAML) on a cluster of workstations are reported in [24] and in Fig. 15. They show that, at least for “synthetic” applications, performances can get very close to hand-written C+MPI code (less than 10 % overhead). Moreover, it turns out that, at least for coarse and medium-grained computation schemes, the mechanism used for handling nesting does not entail a significant performance penalty. Together with those reported by Danelutto in [13], these results confirm the merits of dynamic MDF execution models with respect to template-based ones. SKIPPER [-d] run-time performances could be further improved by integrating some optimization techniques described in [13]. These techniques include a more sophisticated management strategy of the instruction pool (based on high/low water marks), local caching of data on worker nodes and, most noticeably, a distributed interpreter implementation. The current SKIPPER [-d] implementation relies on a centralized data-flow interpreter and a rudimentary scheduling strategy for fireable instructions and is likely not to provide comparable performances in case of very irregular fine-grained computations. Predictability of performances is clearly
harder to obtain than with template-based implementation systems but does not seem an intractable problem (like in SKiPPER [-ii]). The interpretation of profiling results is also easier than with SKiPPER [-ii], especially if sophisticated visualization tools such as jumpshot [28] are provided. The portability of the SKiPPER [-d] runtime system on architectures made of specialized or digital signal processors is currently limited by the fact that it is written in OBJECTIVE CAML and uses bytecode threads. But the runtime could easily be rewritten in C for these systems. In this case, threads can be emulated using hardware context switching mechanisms (as evidenced by the implementation of the SYNDEx kernel for DSPs [18]).

Fig. 16. SKiPPER [-d] performance figures (df skeleton)

4 Comparative assessment

Table 1 summarizes our assessment of the successive versions of SKiPPER. In this table, we have tried to rate each version in terms of the five criteria explicated in Section 3: balance between compile-time and run-time system (Rts/Cts), efficiency (Eff), expressivity (Expr), portability (Port) and predictability (Pred). For this we use a relative “score” between 1 and 4. For the Rts/Cts criteria, 1 means a fully static system – for which all decisions regarding mapping and scheduling of functions are taken at compile-time – and 4 a fully dynamic system for which all these decisions are taken at runtime. For the other criteria 1 means “poor” and 4 “excellent”. The second column recalls the underlying intermediate representation (IR): Synchronous Data Flow Graphs, Parametric Process Networks, Hierarchical Task Graphs and Dynamic Data Flow Graphs.

The evolution from SKiPPER [-o] to SKiPPER [-d] can be viewed as a progressive shift – evidenced by the growing part of the run-time system in the implementation – from static approaches, offering excellent performances and pre-

\[20\] The current implementation is less than 500 lines of OBJECTIVE CAML code. We think that a re-implementation in C would be in the range of 1000-2000 loc, perfectly suited for small memory-print processors.
dictability at the price of a limited expressivity, to more dynamic approaches, trading off efficiency and/or predictability in favor of expressivity.

Fully static approaches, like in SKIPPER [-o], are attractive in our context of embedded reactive applications because they minimize the resources needed to implement the algorithm and allow strict real-time bounds to be computed. But within a programming methodology dedicated to the fast prototyping of solutions – and mainly intended to algorithmicians, not parallel programming specialists – these approaches were finally found too restrictive. For instance, it is often possible to reformulate an existing vision algorithm – defined in terms of dynamic allocated data structures as lists or trees – so that it only uses fixed-size arrays and can be parallelized using a “static” skeleton (like scm); but we found that it is not reasonable, even desirable, to do this reformulation at the prototyping level, when being able to quickly test various algorithmic and/or parallel implementation schemes turned out to be more important than obtaining optimal performances. Moreover, some algorithms are intrinsically not amenable to a static implementation because the size of the input data and/or the duration of the sequential functions cannot be reliably estimated at compile time.

On the other hand, the conclusions given in Section ?? show that approaches relying on a fully dynamic run-time system, like SKIPPER [-ii], may raise efficiency and predictability or observability problems that conflicts with our prototyping goals and/or target platforms (although these approaches might prove useful in other application domains).

In this light, we believe that the SKIPPER [-d] approach offers the best trade-off between the conflicting abovementioned criteria. The data-flow interpretation mechanism is “mostly dynamic” \(^\text{21}\) but its run-time behavior can be more easily modelized and performances do not suffer from hardly understandable performances drops due to unpredictable process allocation \(^\text{22}\).

\(^\text{21}\) Scheduling is done at run-time but mapping of threads to processors is done at compile-time.

\(^\text{22}\) As for the master processes in SKIPPER [-ii].
5 Related work

In the past decade, the issues related to skeleton-based parallel programming have been investigated by several research groups\textsuperscript{23}. But few of have produced full-fledged software environments that can be used to implement complex, realistic applications.

The Pisa Parallel Programming Language (P3L) project is one of this project. The P3L system includes both task parallelism (farm, pipe) and data parallelism (map, reduce, scan). Some control skeletons (loop, seq) allow the definition of sequential P3L modules and the iteration of skeleton compositions. Like Skipper, P3L uses C to express the sequential parts of the application but, unlike it, the skeletal structure of the application is denoted using C-like syntax of data types and skeletons. The first compilers generated code for Transputer-based Meiko CS/1 MIMD machine and for PVM running on a cluster of UNIX workstations. A more recent version (anacleto, [6]) generates C+MPI code for PC running Linux and Fujitsu AP1000. P3L has been used to implement applications such as optical character recognition, ray tracing and circuit test generation (FIND REF ?).

The Heriot-Watt group has investigated the use of skeletal-based methodology for the parallelisation of vision algorithms [20,23,21] CHECK REF. Parallelism is extracted and exploited from programs written entirely in Standard ML. Unlike Skipper or P3L, in which skeletons are viewed as explicit indications to the compiler of where and which parallelism will be deployed – they take an implicit approach, in which skeletons are viewed as possible realizations of common higher-order functions (the decision is taken by the compiler, on the basis of profiling information collected by an instrumentation phase). Results have been given given for a Meiko CS, a Fujitsu AP-1000 and a 32-nodes Beowulf.

The Skil project [?] is another system relying on skeletons to provide a high level parallel programming level. Skil is an imperative, C-based language enhanced with a series of functional features such as higher-order functions and polymorphism. Compile-time instanciation of these features result in very efficient code (approaching the efficiency of direct C implementations). Skil focuses on data-parallelism and provides built-in types for manipulating distributed data-structures. On numerical applications such PDE solvers [?] Skil has demonstrated good absolute performances and scalability (24 speedup for 32 processors, 87 on 128 processors) on 1024-nodes Parsytec multi-processor.

\textsuperscript{23} See for example [29] for a comprehensive survey.
6 Conclusions and future work

Several lessons were learnt when developing and using the SKIPPER system, both at the application level (from a user’s point of view) and at the implementation level (from an implementor’s point of view).

At the application level, the SKIPPER project has provided a convincing demonstration of the merits of skeleton-based parallel programming techniques. These conclusions are supported by realistic case studies, carried out with the help of full-fledged parallel programming environments, by people who were not parallel programming specialists at the first place. First, the “off-the-shelf” style provided by the skeleton approach effectively provides dramatic savings in development effort. These savings make it possible to adopt a truly experimental approach in the design and implementation of applications, a key property in our context. The price to pay is a decrease in performances (compared to hand-crafted parallel code) but, for most of the realizations presented here this can be kept reasonable and was viewed as acceptable, anyway, with regard with the above mentioned benefits. Second, within a given application domain, such as reactive embedded vision, skeletons may be viewed as a very effective way to encapsulate and reuse the expertise gained by skilled parallel programmers. This pragmatically solves the classical “completeness” problem often associated with skeleton-based parallel programming methodologies – namely the fact that, in theory, nothing can guarantee that a given set of skeletons will be sufficient to express every parallel algorithm: in our case, the definition of the skeleton basis was made in a bottom-up manner starting from an identifiable corpus of applications and/or expert knowledge and was explicitly targeted towards low to mid-level vision algorithms. Finally, it could be objected to the explicit, “menu-driven” approach proposed by SKIPPER that it requires a minimum understanding of the skeleton operational semantics to be used and therefore that it cannot be used as fully automatic parallelizing tool. Our answer, motivated by our experience in developing complex vision applications with algorithmicians, is that skeletons actually provides an effective common ground for sharing expertise between image processing and parallel programming specialists: the former no longer have to deal with implementation details and the latter can treat application-specific functions as black boxes.

At the implementation level, the SKIPPER project has led us to thoroughly investigate the relative merits and flaws of static and dynamic approaches for implementing skeletons. As stated in Sec. 3.4, we now believe that a macro data-flow representation of skeleton-based parallel programs is probably the best choice, because it can be associated with a wide spectrum of operational semantics (from purely static synchronous to dynamic tagged-token). This conclusion is similar to the one drawn by Najjar et al in [22]
who underline the “universality” of the data-flow model by exhibiting potential application domains both in the “software” domain (parallel programming on clusters of workstations for instance) and in the “hardware” domain (design of application-specific circuits for instance). In this context, we are now investigating the possibility of developing transformational rules to derive automatically a static formulation of an algorithm (using a synchronous data-flow execution model) from a dynamic one (based upon a tagged-token execution model). Our ultimate goal, motivated by our experience and needs in reactive vision applications, is to be able to specify, with the same skeletal formalism both “hard” (time-critical) parallel applications (built from static skeletons such as SCM) and “softer” applications (built from dynamic skeletons such as DF) which can tolerate the run-time unpredictability implied by interpreter-based implementation techniques. Recent work on graph factorization techniques [14] has provided some insights on how to do this in the context of compile-time bounded iterations. We are currently working to extend this scheme to generic data and task farming skeletons (the fundamental issue being: what constraints do we have to put on the tagged-token data-flow graph formulation of an algorithm — that can always be interpreted dynamically — to make it amenable to static implementation).

References


[29] http://hypatia.dcs.qmw.ac.uk/SEL-HPC/Articles/SkeletonArchive.html